Universidade Federal Fluminense
Instituto de Computação
Redes de Computadores II

Relatório Trabalho Prático, Etapa I

Alunos:
Rafael Tiribás
Victor Patricio

Outubro
2023

Universidade Federal Fluminense
Instituto de Computação
Redes de Computadores II

Relatório

Relatório referente a primeira etapa do trabalho prático da matéria Redes de Computadores II.

Alunos:

Rafael Tiribás

Victor Patricio

Outubro
2023

Conteúdo

1 Introdução

Este trabalho prático tem como objetivo desenvolver uma aplicação de videoconferência descentralizada utilizando comunicação por *sockets*. Os usuários devem poder se registrar no servidor, consultar a lista de cadastrados e se conectarem aos seus pares utilizando o modelo Peer-to-Peer (P2P). Na etapa correspondente a este relatório, foi implementado um *socket* TCP que interconecta os clientes com o servidor e fornece as funções básicas de operação do sistema.

1.1 Sockets

Para estebelecer conexões entre processos em um rede de computadores utilizamos os mecânismos *sockets*. Assim, em uma conexão bidirecional conseguimos prover a comunicação entre duas pontas(processo cliente e processo servidor) que estejam na mesma rede. Podendo ser utilizado quando a aplicação necessita realizar envio de arquivos, troca de mensagens, ou qualquer ação que envolva a transferência de recursos. Basta que o processo cliente estabeleça uma conexão com o processo servidor, permitindo assim a comunicação entre ambos.

1.2 Protocolo

Para a implementação dos sockets podemos optar por dois protocolos diferentes: TCP e UDP. O primeiro é utilizado quando a conexão estabelicida requisita confiabilidade na sua aplicação. Já o protocolo UDP, prioriza a alta velocidade de transmissão e baixa latência em detrimento da confiabilidade dos dados transmitidos. Para a aplicação do Trabalho Prático foi utilizado o protocolo TCP.

1.3 Threads

A biblioteca *threading* permite a criação e gerenciamento de *threads* em *Python*. As *threads* permitem que um programa execute múltiplas tarefas concorrentemente, ideal para operações que podem ser executadas de forma independente. Neste caso, foi utilizado para que o processo servidor lide com cada conexão de cliente de maneira simultânea, sem que haja interrupções ou que um processo tenha que aguardar o término de outro para ser executado.

2 Implementação

A Etapa I do Trabalho Prático foi implementado em um ambiente *Windows* com a linguagem de programação *Python* na versão 3.12.0 instalada. O código está dividido em dois arquivos: servidor.py e client.py.

2.1 Servidor

Primeiramente importamos as bibliotecas necessárias(*socket* e *threading*) utilizando o comando *import*. Em seguida definimos os dados do servidor que serão utilizados na criação do *socket*. Como a porta, o endereço IP e o formato da mensagem que será trocada entre os processos. Com as váriáveis definidas, declaramos enfim o *socket* passando como parâmetro o protocolo que será utilizado, neste caso o TCP, e depois associamos ele ao endereço do servidor definido anteriormente.

```python
import socket
import threading

HEADER = 64
PORT = 5050 # Porta do servidor.
SERVER = socket.gethostbyname(socket.gethostname()) # Pega o IP da máquina automaticamente.
ADDR = (SERVER, PORT)
FORMAT = 'utf-8'

#   Socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(ADDR)
```

Figura 1: Início do código do processo servidor

Utilizamos a estrutura dicionário do *Python* para armazenar uma tabela dinâmica contendo as informações dos clientes. Essa escolha foi tomada pois assim conseguimos associar de maneira eficiente o nome de usuário(Chave) com o seu endereço(Valor). Com efeito, a consulta de usuários por nome foi otimizada. Além de ter facilitado a operação de desvincular um cliente da aplicação quando for solicitado.

A função *iniciar()* foi definida para dar início ao funcionamento do processo servidor. Nela, o servidor é colocado para "escutar"possíveis conexões dentro de um *loop*, que só é encerrado quando a execução deve ser interrompida. Ao receber uma nova conexão, o código declara uma *thread* para executar a função de gerenciamento do cliente. Dessa forma, o servidor consegue lidar com múltiplas comunicações de maneira simultânea.

Figura 2: Começo da execução

No gerenciamento do cliente, é feito a decodificação da mensagem e identificação da requisição. Três requisições são definidas: cadastro, consulta e desconexão. Para as duas primeiras, é passado como parâmetro das funções específicas à elas o nome de usuário a ser cadastrado ou consultado. Já na terceira, o código apenas finaliza a conexão e remove o usuário da tabela de clientes.

```python
#   Função para lidar com o processo cliente e suas requisições.
def gerencia_cliente(conn: any, end: any) -> None:
    print(f"[Nova Conexão] {end} conectado.")
    conectado = True
    while conectado:
        tamanho_msg = get_tamanho(conn)
        if tamanho_msg:
            msg = conn.recv(tamanho_msg).decode(FORMAT)
            msg = msg.split()
            match msg[0]:
                case "CADASTRO":
                    nome = msg[1]
                    cadastro(nome, end, conn)
                case "CONSULTA":
                    endereco = consulta(msg[1])
                    conn.send(f"[ENDERECO {msg[1]}]: {endereco}".encode(FORMAT))
                case "DESCONECTAR":
                    conectado = False
                    remove(nome)
                    conn.send("[DESCONECTADO]".encode(FORMAT))
            print(f"[{end}] {msg}\n[TABELA USUÁRIOS ATIVOS] {usuarios}")
    conn.close()
```

Figura 3: Gerenciamento de clientes

Antes de cadastrar um novo usuário, é verificado se o nome fornecido pelo cliente já não consta na tabela dinâmica do servidor. Caso não esteja, o cliente é inserido no dicionário. Utilizando seu nome como chave e o IP/Porta como valor. O sucesso ou não do cadastro é retornado para o cliente com uma mensagem no terminal.

```python
#   Função de cadastro.
def cadastro(nome: str, endereco: str, conn: any) -> None:
    if usuarios.get(nome, 0) == 0:  # Checa se o nome já não está cadastrado no sistema.
        usuarios[nome] = endereco
        conn.send("[CADASTRO REALIZADO COM SUCESSO]".encode(FORMAT))
        return
    conn.send("[ESTE USUÁRIO JÁ ESTÁ CADASTRADO]".encode(FORMAT))
```

Figura 4: Cadastro de usuário

Como optamos por utilizar dicionários, as operações de consulta e remoção foram extremamente facilitadas. Sendo necessário apenas dar um *get* com o nome do usuário para obter seu endereço ou um *pop* para remove-lo da base de dados.

```python
#   Função de consulta do cliente.
def consulta(nome: str) -> str:
    return usuarios.get(nome, 0)

#   Função de remoção de um usuário.
def remove(nome: str) -> None:
    usuarios.pop(nome)
```

Figura 5: Consulta e remoção

2.2 Cliente

Assim como no processo servidor, começamos importanto a biblioteca *socket* no código. Em seguida definimos os dados do servidor a ser conectado. Incluindo a porta, o endereço IP e o formato da mensagem que será trocada entre ambos. Por fim, é declarado o *socket* do cliente especificando o protocolo TCP e feita a conexão dele com o endereço do servidor.

```python
import socket

HEADER = 64
PORT = 5050 # Porta do servidor.
SERVER = socket.gethostbyname(socket.gethostname())
FORMAT = 'utf-8'
ADDR = (SERVER, PORT)

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(ADDR)
```

Figura 6: Início do processo cliente

Logo após, é declarada uma função para decodificar a mensagem do cliente para o formato especificado da conexão e enviá-la. A mensagem enviada pela função é uma *string* resultado da concatenação da *string* identificadora de requisição com o nome de consulta ou cadastro de usuário. No caso da requisição ser de desconexão a *string* é apenas a identificadora.

```
def envia(msg):
    mensagem = msg.encode(FORMAT)
    tamanho_msg = len(mensagem)
    envia_tamanho = str(tamanho_msg).encode(FORMAT)
    envia_tamanho += b' ' * (HEADER - len(envia_tamanho))
    client.send(envia_tamanho)
    client.send(mensagem)
```

Figura 7: Função de envio

Por fim, a execução entra em um loop que gerencia as requisições do usuário. Este loop é encerrado quando o usuário deseja encerrar sua conexão com o servidor.

```
conectado = True
while conectado:
    print("[CADASTRO]   |   [CONSULTA]    |   [DESCONECTAR]")
    opcao = input()
    match opcao:
        case "CADASTRO":
            print("[DIGITE O NOME DE USUÁRIO]:")
            nome = input()
            envia(f"{opcao} {nome}")
            print(client.recv(2048))
        case "CONSULTA":
            print("[DIGITE O NOME DE USUÁRIO DO ENDEREÇO A SER CONSULTADO]:")
            nome = input()
            envia(f"{opcao} {nome}")
            print(client.recv(2048))
        case "DESCONECTAR":
            print("[VOCÊ SERÁ DESCONECTADO E DESVINCULADO DO SERVIDOR DE REGISTRO].")
            envia("DESCONECTAR")
            conectado = False
            print(client.recv(2048))
```

Figura 8: Função de envio

3 Execução

A execução desta aplicação pode ser feita de maneira simples. Inicialmente, deve-se executar o programa *servidor.py*. Para executá-lo basta um simples comando no terminal do *Windows*: *pyhton servidor.py*. Agora com o servidor iniciado, já é possível executar o processo cliente e estabelecer a comunicação entre ambos. Sua execução é idêntica a do servidor, apenas mudando o nome do arquivo. É possível executar quantos processos clientes for desejado. O terminal do cliente indica as possíveis operações a seres executadas, bastando apenas inserir no terminal um dos comandos de requisição: CADASTRO, CONSULTA e DESCONECTAR. Todos os comandos e requisições do sistema são impressos no terminal de maneira intuitiva, tanto no do servidor quanto nos dos clientes.

4 Conclusão

Todo o desenvolvimento do trabalho, incluindo o relatório, foi feito de maneira conjunta. Sendo realizadas reuniões semanais do grupo para debater ideias e discutir os resultados obtidos. Além disso, ambos os participantes produziram a implementação da Etapa I, cada um de sua maneira. Dessa forma, o resultado final do trabalho foi uma junção do que deu certo em cada implementação e as soluções que cada participante propôs para os problemas que surgiram na execução desta etapa. Portanto, é válido concluir que todos os participantes tiveram igual participação no resultado final do trabalho. Todos os códigos desenvolvidos estão disponíveis em repositórios públicos do *Github* de cada participante.